VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Implementace transformací modelů postavených na Petriho sítích v nástroji Kaira

Implementation of Transformations of Petri Nets Based Models in a Tool Kaira

2014 Martin Kozubek

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Martin Kozubek**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Implementace transformací modelů postavených na Petriho sítích v nástroji Kaira
Implementation of Transformations of Petri Nets Based Models in a Tool Kaira

Zásady pro vypracování:

Na katedře je vyvíjen nástroj Kaira. Tento nástroj je určen pro modelování paralelních – distribuovaných aplikací pomocí barevných Petriho sítí. Z těchto modelů pak Kaira umožňuje generovat samostatné aplikace v jazyce C++, které využívají vláken a MPI. Hlavním cílem bakalářské práce bude rozšířit tento nástroj o transformace. Základní myšlenkou je umožnit ve fázi modelování použit konstrukce na vyšší úrovni abstrakce (jako moduly, popsáno v doporučené literatuře), které pak jsou automaticky transformovány na základní konstrukce nástroje Kaira. Cíle bakalářské práce lze shrnout v těchto bodech.
1. Seznamte se s nástrojem Kaira a aktuálním stavem jeho vývoje.
2. Rozšiřte nástroj Kaira o podporu transformací. Při implementaci tohoto rozšíření spolupracujte s Lukášem Tomaszkem.
3. Navrhněte vhodné transformace a tyto transformace realizujte.
4. Na praktických příkladech demonstrujte funkčnost řešení.

Seznam doporučené odborné literatury:

Stanislav Böhm and Marek Běhálek. 2012. Usage of petri nets for high performance computing. In Proceedings of the 1st ACM SIGPLAN workshop on Functional high-performance computing (FHPC '12). ACM, New York, NY, USA, 37-48. DOI=10.1145/2364474.2364481 http://doi.acm.org/10.1145/2364474.2364481

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Marek Běhálek, Ph.D.**

Datum zadání: 16.11.2012

Datum odevzdání: 07.05.2014

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 5. května 2014 ..... Kozubek .....

Abstrakt

Kaira je vývojové prostředí určené k modelování paralelních-distribuovaných aplikací. Modelování je postaveno na barevných Petriho sítích a z těchto modelů umí Kaira vygenerovat aplikaci v jazyce C++ využívající MPI. Hlavní náplní tohoto textu je rozšíření modelu o novou konstrukci, umožňující vkládání podsítě(modulu) do přechodu v hlavní síti. Nástroj bude model automaticky transformovat, tak aby obsahoval pouze základní konstrukční prvky a dodržel přitom požadované chování.

Klíčová slova: paralelní programování, Kaira, barevné Petriho sítě, modelování, automatická transformace

Abstract

Kaira is a development tool designed for modelling of parallel-distributed applications. Models are based on Coloured Petri Nets. Kaira is able to generate C++, MPI application based on these models. The main scope of this text is expanding the model by new structure, allowing insertion of the subnet(module) to the transition in the main net. The tool will automatically transform the model so that it only contains the basic structural elements, but that it keeps the desired behavior.

Keywords: concurrent programming, Kaira, Coloured Petri Nets, modelling, automatic transformation

Obsah

Seznam obrázků

1 Úvod

Na katedře je vyvíjen nástroj Kaira určený pro tvorbu paralelních-distribuovaných aplikací. Kaira je vyvíjena jako kompletní programovací prostředí s vysokoúrovňovým přístupem k paralelnímu programování. K popisu paralelního chování se využívá grafických modelů postavených na formalismu barevných Petriho sítí[1]. Do modelu může být integrován vlastní kód v jazyce C++. Na základě těchto modelů Kaira generuje program využívající technologii MPI[2]. Kaira je Open Source projekt volně ke stažení na jeho webových stránkách[3].

Základními prvky modelu popisujícího paralelní chování jsou místa, přechody a orientované hrany, které vedou vždy jen mezi přechodem a místem. Místa reprezentují paměťový prostor a mají zadaný svůj datový typ. Přechody popisují manipulaci s daty v místech včetně distribuce dat mezi výpočetními jádry a může do nich být integrován zdrojový kód.

Tento text pojednává o rozšíření modelu o novou konstrukci umožňující vkládat do přechodů moduly. Modulem rozumíme již hotový model v Kaiře, který obvykle plní nějakou funkci a může z něj být vygenerovaný i samostatný kód. Řešení bylo implementováno již dříve takovým způsobem, že se vygeneroval kód pro modul i hlavní model a hlavní program pak vytvářel a ukončoval instance modulu za běhu. Řešení je popsáno v [4], ale v pozdější verzi bylo z Kairy odstraněno, protože neumožňovalo dostatečnou kontrolu nad rozložením zátěže mezi výpočetní uzly. Nové řešení bude využíváno na úrovni modelování a bude se snažit poskytnout uživateli maximální kontrolu a komfort při tvorbě složitějších programů. Jádrem řešení je transformovat hlavní model i modul tak, aby vznikl jeden model, který bude obsahovat pouze základní konstrukční prvky.

Detailnější popis Kairy, Petriho sítí a MPI lze nalézt v následující kapitole. Vkládání modulu do přechodu je věnována kapitola 3. První část kapitoly 3 obsahuje širší úvod do problematiky modulů a souvisejícího uživatelského rozhraní. V druhé části rozeberu základní problémy, které je třeba při automatické transformaci řešit, jako je inicializace míst nebo umístění nově vložených prvků. Kapitola 4 obsahuje několik technických detailů a třídní diagram s třídami a metodami, kterých se týká implementace. V kapitole 5 představím příklad modelu násobení tří matic s využitím modulu řešícího paralelní násobení dvou matic a v předposlední kapitole popíši další možnosti transformace.

2 Nástroj Kaira a další použité technologie

2.1 Motivace a cíle nástroje

V prostředí vědeckých a technických výpočtů se běžným nástrojem stávají paralelní počítače. Umožňují nám snížit výpočetní čas a nebo spouštět aplikace se strukturou příliš rozsáhlou pro jediný stroj. Vývoj aplikací pro paralelní systémy je však mnohdy složitý, a to ve všech vývojových fázích.

Nástroj Kaira je vyvíjen jako kompletní programovací prostředí určené k vytváření paralelních-distribuovaných aplikací. Zaměřuje se na použití v oblasti výpočetních klastrů a superpočítačů o desítkách až stovkách výpočetních uzlů. Cíle pro tento nástroj, jak je deklaroval hlavní vývojář, jsou následující:

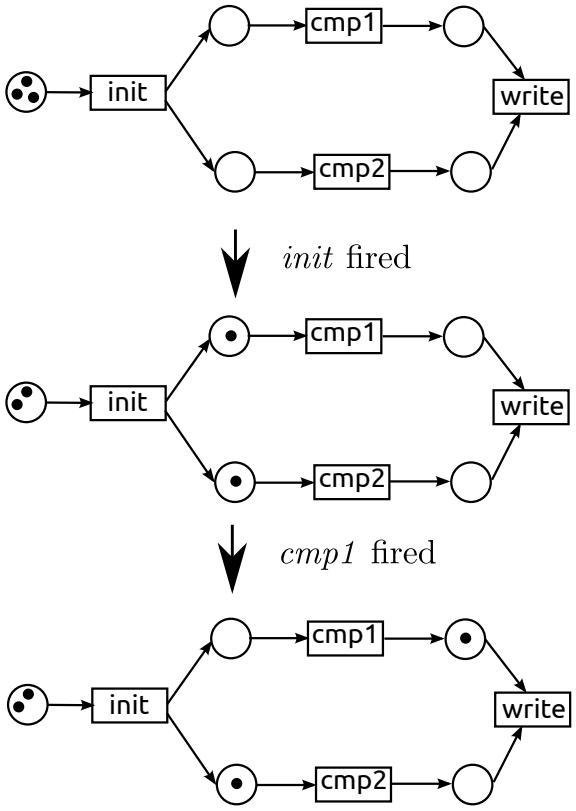
- *Abstraktní výpočetní model* - Paralelismus v aplikaci není Kairou sám detekován, ale uživatel jej definuje pomocí grafického modelu. Hlavním požadavkem na model je, aby byl dostatečně abstraktní, aby mohl nástroj být používán i programátory, kteří nejsou experty v oboru paralelního programování, a to bez znalosti nízkoúrovňových technologií.
- *Integrace s existujícími zdrojovými kódy* - Do modelu lze integrovat vlastní sekvenční kód
- *Rychlé prototypování* - Vývoj distribuovaných aplikací je komplexní a často zdlouhavý, a proto chceme mít k dispozici funkční prototypy částí aplikace už na počátcích vývoje
- *Rozumě rychlý výsledný program* - Pro použití v oblasti rozsáhlých vědeckých a technických výpočtů záleží také na rychlosti výsledného programu

Paralelní stránky a komunikace ve vytvářených programech jsou určovány grafickým modelem, do kterého lze zakomponovat části kódu. Na základě grafického modelu, částí kódu a přiložených knihoven bude vygenerováno paralelní řešení. Základem pro modelování v Kaiře jsou Petriho sítě, a proto budou představeny v následující kapitole.

2.2 Petriho sítě

Petriho sítě jsou modelovací vizuální nástroj vhodný k modelování distribuovaných systémů. Jednou ze základních forem jsou P/T petriho sítě (Place/Transition Petri Nets). P/T Petriho sítě lze chápat jako bipartitní graf tvořený z míst(place), přechodů(transtion) a hran. Místa jsou značena kolečky a přechody obdélníky. Hrany jsou orientované a můžou být vedeny pouze od místa k přechodu nebo od přechodu k místu, nikoli však mezi dvěma místy nebo dvěma přechody.

Místa si lze vyložit jako paměťový prostor, ve kterém se mohou nacházet tokeny. Tokeny jsou v grafu obvykle reprezentovány malými tečkami v místě. Počet tokenů určuje stav místa. K manipulaci s tokeny slouží přechody.



Obrázek 1: Příklad vývoje P/T Petriho sítě

Přechod může být ve dvou stavech, buď proveditelný nebo neproveditelný. Má-li přechod na každém jeho vstupním místě k dispozici token, je proveditelný a může být proveden. Neproveditelným je přechod ve všech ostatních případech. V případě, že dojde k provedení přechodu je z každého vstupního místa token odebrán a na každé výstupní místo je token vložen.

Obrázek 1 ukazuje vývoj Petriho sítě znázorňující jednoduché rozdělení výpočtu do dvou částí. V původním stavu je naplněno pouze jedno místo a proveditelný je přechod *init*. Po provedení přechodu *init* jsou proveditelné přechody *cmp1* i *cmp2* současně. Až se provedou oba dva výpočty bude provedením přechodu *write* zapsán výsledek. Přechody se můžou provádět v jakém–koliv pořadí a v libovolném čase. Tímto způsobem Petriho sítě přirozeně popisují konkurenční a paralelní chování.

P/T sítě jsou považovány za nízkoúrovňový nástroj a modelování problémů reálného světa často směřuje k příliš rozsáhlým sítím. Z tohoto důvodu je sémantika Petriho síti pro využití v různých aplikacích rozšiřována. Jedním takovým rozšířením jsou barevné Petriho sítě [1], jejichž základní myšlenkou je, že tokeny v místech nebudou považovány za anonymní, ale můžou obsahovat různé hodnoty. Každé místo pak má zadaný typ(barvu) a může obsahovat pouze hodnoty tohoto typu. Hodnoty tokenů nám umožňují definovat přesnější podmínky kdy je přechod proveditelný a jaké budou výstupní tokeny produko-

vané při provedení přechodu. Na hranách jsou zavedeny výrazy, které rovněž specifikují chování přechodu. V nástroji Kaira používáme k modelování paralelního chování právě rozšířené barevné Petriho sítě. Podrobnější popis Petriho síťí je například v [5, 6].

2.3 MPI

MPI (Message Passing Interface)[2] je specifikace knihoven určených k zasílání zpráv mezi procesy dostupná v jazyce C, C++ a Fortran a její implementace jsou volně dostupné na internetu. Využívá programovací model, kdy je jedna aplikace spouštěna na více procesorech a obsahuje logiku k rozdělení práce. Je určena pro systémy s distribuovanou pamětí. Systémy s distribuovanou pamětí jsou obvykle implementovány jako počítače s nějakou formou propojení, kde každý počítač se skládá z procesoru a vlastní paměti. Výpočetní úlohy jednotlivých procesorů můžou probíhat pouze nad daty v jejich vlastní paměti a pokud jsou vyžadována data ze vzdálené paměti, musí procesory mezi sebou komunikovat. Při využití MPI probíhá komunikace mezi procesory zasíláním zpráv jako na následující ukázce.

```
#include<mpi.h>
int main() {
    char data [10];
    MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
    if (myrank == 0){
        fill_data_with_something (data);
        MPI_Send(data, 10, MPI_CHAR, 1, 1, MPI_COMM_WORLD)
    }
    if (myrank == 1){
        MPI_Recv(data, MPI_CHAR, 10, MPI_ANY_SOURCE, MPI_ANY_TAG, MPI_COM_WORLD)
        do_something(data);
    }
    MPI_Finalize();
    return 0;
}
```

Výpis 1: Posílaní zpráv s využitím MPI

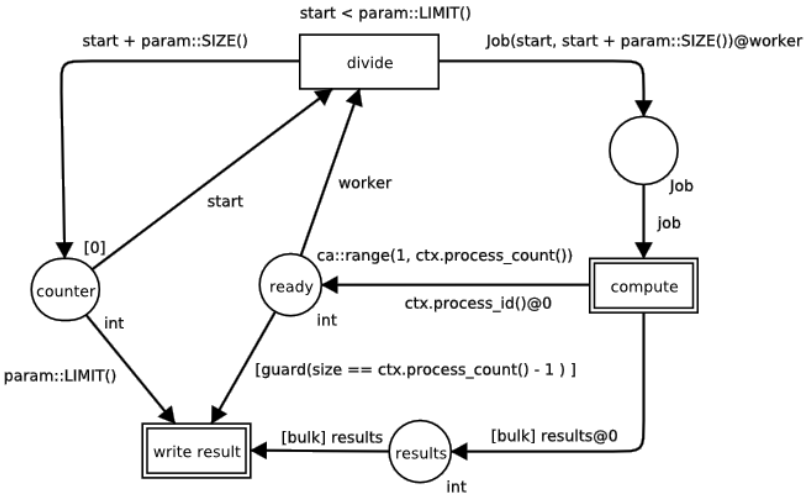
Výpis kódu 1 ukazuje základní konstrukce MPI. Při spuštění tohoto programu začne několik procesů vykonávat funkci main. Procesy běží nezávisle na sobě v odděleném paměťovém prostoru, a jediný způsob jak mezi sebou mohou komunikovat je pomocí zasílání zpráv. V prvním kroku funkcí `MPIComm_rank` zjistíme označení procesu. Proces s označením 0 vytvoří ve své paměti nějakou zprávu, v tomto případě deseti–znakové slovo, a pošle ji procesu 1 s tagem 1. Proces 1 čeká na zprávu od jakéhokoliv zdroje s jakýmkoliv tagem, následně data příjme a zpracuje. MPI umožňuje pracovat s většinu základních datových typů jazyka C++. Obsahuje také mnoho dalších funkcí, například pro broadcasting nebo shromažďování dat od všech procesů.

Výhodou MPI je, že při implementaci byl kladen důraz především na výkon, škálovatelnost a přenositelnost programu. Na druhou stranu MPI prosazuje nízkoúrovňový přístup a vytvoření paralelní verze aplikace obvykle trvá delší dobu. Kaira však prosazuje vysokoúrovňový přístup, kdy se paralelní chování znázorňuje grafickým modelem a

MPI kód je vygenerován podle modelu automaticky. Navíc lze části programu velmi brzy otestovat a znalost MPI není nutná.

2.4 Popis funkcí nástroje

Vývoj Kairy je v současné době zaměřen na optimalizace, profilování a integraci s existujícími programy[4]. Nicméně implementace Kairy je ve stavu, kdy jsou všechny nutné části funkční a lze vytvářet a testovat reálné programy. Funkčnost nástroje proto můžeme demonstrovat na příkladě. V paralelním programování je běžný případ, kdy jeden hlavní proces (master) rozděluje úlohu mezi dílčí procesory (worker). Když worker úlohu dokončí, odešle výsledky masterovi a hlásí se o novou úlohu. Obrázek 2 obsahuje síť řešící tento problém.



Obrázek 2: Model rozdělení úlohy metodou master-workers

2.4.1 Model

Grafická reprezentace a sémantika modelu jsou založeny na Barvených Petriho sítích[1]. Barvou vrcholu tady rozumíme datový typ místa, který je napsán pod místem vpravo dole. Jako datové typy lze použít všechny základní C++ typy a nebo lze vytvořit vlastní datový typ, u kterého je ale potřeba definovat funkce jako je konstruktor a serializace pro zasílání ve zprávě. Příkladem vlastního datového typu je na obrázku Job. Na hranách lze používat výrazy jazyka C++. Často je na hraně pouze identifikátor proměnné, ale lze volat i funkci. Navíc lze použít výrazy v hranatých závorkách `bulk` a `guard`. Za guard lze napsat do závorek podmínku a bulk říká, že se bude pracovat se všemi dostupnými tokeny. Přechody také umožňují použít guard, který je napsaný nad přechodem. Nově

lze přechodu zadat také prioritu a tím ovlivnit pořadí provedení přechodu v případě, že jich je proveditelných více.

Největší rozdíl v porovnání s Petriho sítěmi je v předpokladu, že při běhu aplikace bude existovat několik kopií sítě současně. Podle jejich významu kopie sítě nazýváme net–procesy. V praxi poběží jedna kopie sítě na jeden výpočetní uzel a v simulaci si počet net–procesů můžeme zvolit. Každý net–proces je samostatný, může mít odlišné značení a poběží nezávisle na ostatních. Jednotlivé net–procesy spolu mohou komunikovat jediným způsobem. Provedený přechod může vytvořit tokeny v jiném net–procesu, určeném v modelu výrazem za značkou @. Tato sémantika jde ruku v ruce s podstatou distribuované paměti, kde proces nemůže číst z paměti jiného procesu, ale může odesílat zprávy.

Vratmě se k příkladu master-workers (obr. 2). Při běhu programu bude existovat několik kopií sítě. V modelu je identifikujeme číslem začínajícím od 0 a počet net-procesů lze zjistit funkcí `ctx.process_count()`. Když se program spustí, tak se daty inicializují pouze místa na net–procesu 0, a proto bude tento proces obsazen v roli master. Místo *ready* reprezentuje volné workery, tudíž bude na začátku v procesu 0 obsahovat čísla `1, 2, ..., ctx.process_count()-1` . V ostatních procesech budou všechna místa prázdná.

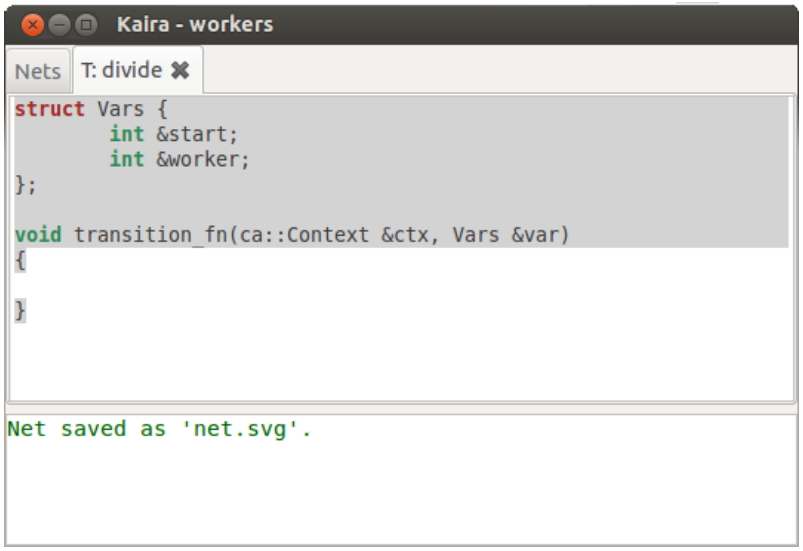
Výpočetní úloha v tomto příkladě představuje výpočet dat nad intervalem. Parametr #SIZE udává velikost jednoho intervalu a #LIMIT je maximum, při kterém výpočet skončí. Místo *counter* udržuje začátek intervalu. Přechod *divide* přiděluje workerům intervaly. Výraz `(start, start + #SIZE)@worker` vytvoří token typu `(int,int)` reprezentující interval v procesu s číslem workera. Samotný výpočet nad intervalem je reprezentován přechodem *compute*. Dvojitý rámeček znamená, že je uvnitř přechodu vložena funkce. Potom co worker provede výpočet, odešle výsledky zpět masterovi na proces 0 a zároveň vytvoří token s jeho id procesu v místě *ready*. Výpočet končí když *counter* dosáhne limitu a všichni workeři dokončili výpočet, tedy v místě *ready* je stejný počet tokenů jako na začátku.

2.4.2 Integrace kódu

Důležitou vlastností Kairy je propojování vizuálního modelu s existujícími sekvenčními kódy. Vytváření sekvenčních částí aplikace je často jednodušší s využitím běžného programovacího jazyka, zatím co visuální model je využíván k popisu paralelního a distribuovaného chování. Programátor může mít úlohu již vyřešenou ve formě sekvenčních knihoven a nástroj Kaira použít k vytvoření řešení pro paralelní počítače.

První možností jak integrovat kód do modelu je vložení kódu do míst nebo přechodů. Nástroj vygeneruje šablonu na základě vstupních a výstupních hran. Například pokud chceme vložit kód do přechodu *divide*(obr. 2) nástroj otevře textový editor se šablonou jako na obrázku 3. Uživatel může vložit kód jako tělo funkce.

Další možnost jak do projektu vložit kód je přidání nových funkcí, typů nebo knihoven do projektu. Nástroj umožňuje vyplnit hlavičkový soubor v jazyce C++. Všechny knihovny, funkce, struktury a objekty z hlavičkového souboru, lze poté používat v modelu. Jedinou podmínkou je, aby datový typ, který používáme v modelu měl definované metody k jeho serializaci a zobrazení při simulaci.



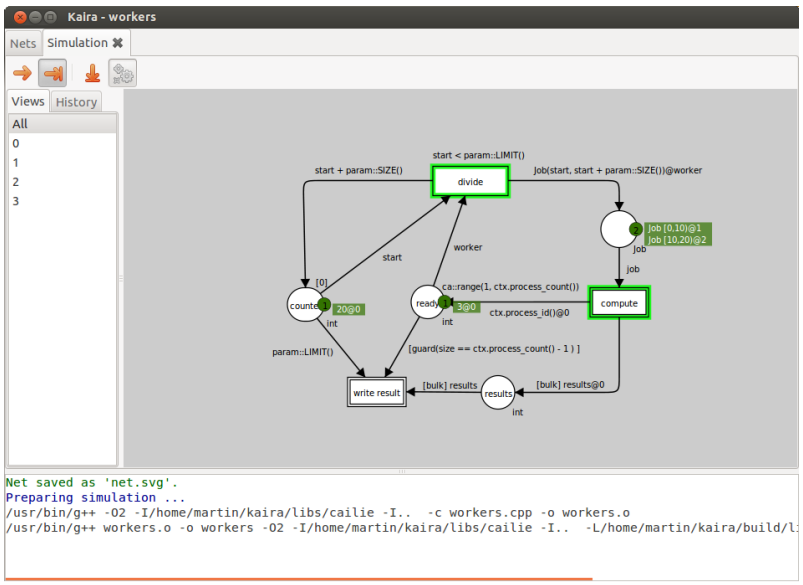
Obrázek 3: Šablona pro kód uvnitř přechodu

2.4.3 Simulace

Druhou důležitou vlastností Kairy je možnost okamžitě vytvořit prototyp aplikace na základě vizuálního modelu a vyzkoušet jeho chování prostřednictvím simulace na stejném stroji. Obrázek 4 ukazuje simulaci nad modelem z obr. 2. V průběhu simulace je vytvořeno zadaný počet kopií sítě, které napodobují běh na paralelním stroji s distribuovanou architekturou. Uživatel si může zobrazit stav libovolné kopie sítě zvlášť a nebo stav všech kopií v jedné síti. Evoluce sítě probíhá na základě přechodů, které se provedou klikem uživatele Musíme však mít na paměti, že v praxi můžou být přechody odpáleny v náhodném čase a libovolném pořadí. Proveditelné přechody jsou zobrazeny zeleně a uživatel je může provézt. Data jsou reprezentovány textovou formou napravo od míst. Číslo za zavináčem nás informuje, ve kterém net-procesu jsou data k dispozici, tedy který proces je má ve své lokální paměti.

2.4.4 Výstup

V současném stavu umí Kaira využívat pouze jazyk C++ a knihovny integrované s tímto jazykem, jako je Octave [9]. Výstupem nástroje je tedy program využívající všechny importovatné a předdefinované knihovny a technologii MPI. Jádrem programu jsou datové struktury reprezentující místa a fronty, které udržují proveditelné přechody.



Obrázek 4: simulace

3 Vkládání podsítě do přechodu

Kaira je nástroj určený pro modelování paralelních distribuovaných aplikací a následné generování samostatného kódu. Modelování je založeno na Barvených Petriho sítích, jenž mají striktní sémantiku založenou na matematickém podkladu. Základní konstrukci grafického modelu v Kaiře tedy tvoří přechody, místa a hrany mezi přechodem a místem (kapitola 2.2). Hlavním cílem této práce je umožnit ve fázi modelování použít konstrukce na vyšší úrovní abstrakce, jako jsou moduly, které jsou pak automaticky transformovány na základní konstrukce nástroje Kaira.

Modulem v Kaiře rozumíme síť, jejíž vygenerovaný kód bude funkce, a která obvykle bývá součástí většího projektu. Síť má označená místa pro vstup a výstup. Výstupem funkce budou data, která po evoluci sítě zůstanou v místě označeném jako výstup. Vstupní místa obvykle nejsou v modelu inicializována, protože budou zaplněna až daty předanými jako argument už vygenerovaného programu. Moduly lze buildovat samostatně, ale jejich chování je identické s chováním přechodu. Modul, stejně tak přechod jsou funkce, které na základě vstupních dat(tokenů ze vstupních míst) vygenerují výstupní data (tokeny do výstupních míst), a proto si vkládání modulu do přechodu, nebo lépe řečeno substituce přechodu podsítí (podobně jako u Hierarchických petriho sítí [7, 1]) mohlo najít své uplatnění.

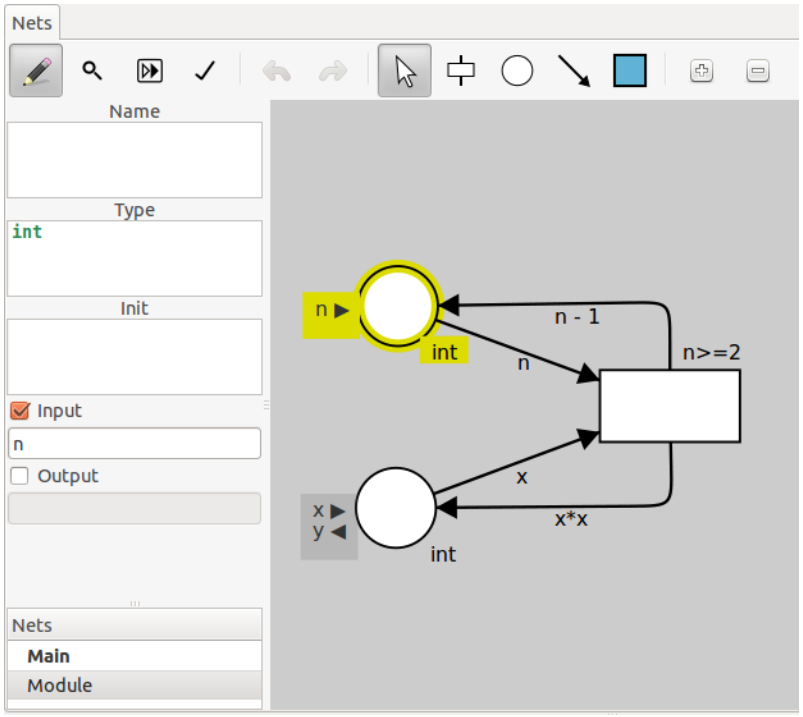
Myšlenka vkládání modulu do přechodu byla realizována již dříve takovým způsobem, že když se měl provést přechod označený jako modul, vytvořila se nová samostatná instance modulu s argumenty přebranými ze vstupních míst. Po ukončení sítě se převzala výstupní data a instance zanikla (více v [4]).

Nové řešení bude aplikováno pouze v oblasti grafického rozhraní a modelu. Model má být rozšířen o podporu vkládání modulu do přechodu a případné určení parametrů transformace. Transformace sítě proběhne automaticky před spuštěním komponenty pro generování kódu. Po transformaci bude model obsahovat opět pouze základní konstrukce nástroje, místa, přechody a hrany. Protože modul je také síť a vkládáme ho do hlavní sítě, používám pro modul také výraz podsíť.

3.1 Uživatelské rozhraní

Tato část textu má za cíl ukázat, jak se bude využívat nové rozšíření, o kterém je celá kapitola. Představuje postup a stěžejní prvky v uživatelském rozhraní.

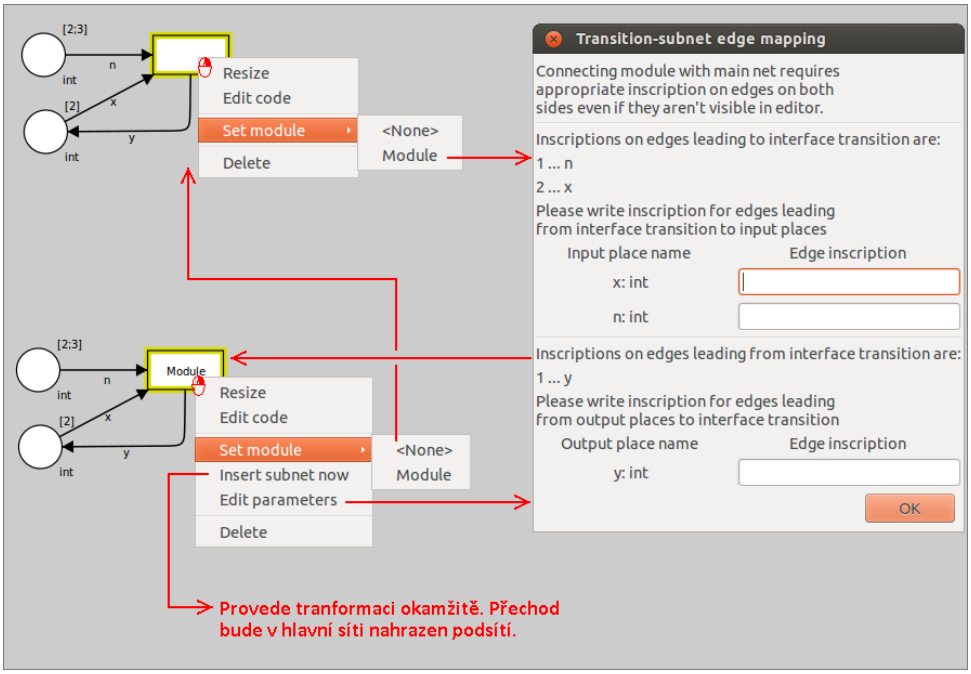
Prvním krokem při vkládání modulu do přechodu je přidání nové sítě do projektu. Novou síť lze do projektu přidat kliknutím pravým tlačítkem myši do oblasti `Nets` v levém sloupci. Hlavní nabídka umožňuje také importovat do projektu sítě z jiných projektů a tím znovu využít již vytvořené modely. Aby se ze sítě stal modul, je třeba označit vstupní a výstupní místa (obr. 5). Vstupní a výstupní místa jsou označena šipkou s názvem vedle ní. Název je zde důležitý především k rozlišení jednotlivých míst při pozdějším využití modulu. Měl by napovídat o jaký vstup nebo výstup se jedná a musí být pro každé místo unikátní. Jak je vidět na obrázku 5 místo může být považováno za vstupní i výstupní místo najednou. Před evolucí se místo inicializuje daty a po evoluci sítě se ze stejného místa data převezmou.



Obrázek 5: Označení vstupů a výstupů

Z hotového modulu můžeme přímo vygenerovat knihovnu pomocí nabídky `Build library`. Kód vygenerovaný z modulu však nemůžeme rovnou nahrát do paměti a spustit. Je to funkce, kterou musíme někde zavolat a obvykle je součástí nějakého většího projektu. V rámci této práce budou moduly využity trochu jinak. Stanou se součástí modelu v Kaiře ještě před vygenerováním kódu a měli by se vůči nadstavené síti chovat stejně jako jeden přechod.

Obrázek 6 ukazuje prototyp rozšířené nabídky přechodu při pravém kliknutí myší sloužící právě pro vložení modulu do přechodu. První co v nabídce přibylo je možnost `Set module`. V nabídce `Set module` se objeví všechny sítě z projektu které mají alespoň jedno označené vstupní místo a alespoň jedno označené výstupní místo. Nabídka rovněž obsahuje možnost `<None>`, která přechod vrátí do původního stavu, kdy na něj editor hledí jako na nově přidaný obyčejný přechod. Po vybrání modulu z nabídky se vždy automaticky otevře dialog jako naznačují na obrázku 6 červené šipky. Tyto šipky ukazují jak se změní zobrazení po vybrání té možnosti, odkud vedou. Na obrázku je zatím prototyp dialogu, který zde slouží pro budoucí napojení podsítě na hlavní síť. Je vyžadována interakce uživatele, protože v případě, že modul bude obsahovat více vstupů nelze automaticky rozhodnout jak napojení provézt. Dialog obsahuje jen tuto část, ale pro případ, že se v budoucnu realizuje některá další nadstavba (kapitola 6), která bude vyžadovat zadání parametrů uživatelem, můžeme si v dialogu představit i tlačítko `Next` pro určení



Obrázek 6: Uživatelské rozhraní pro práci s podsítí

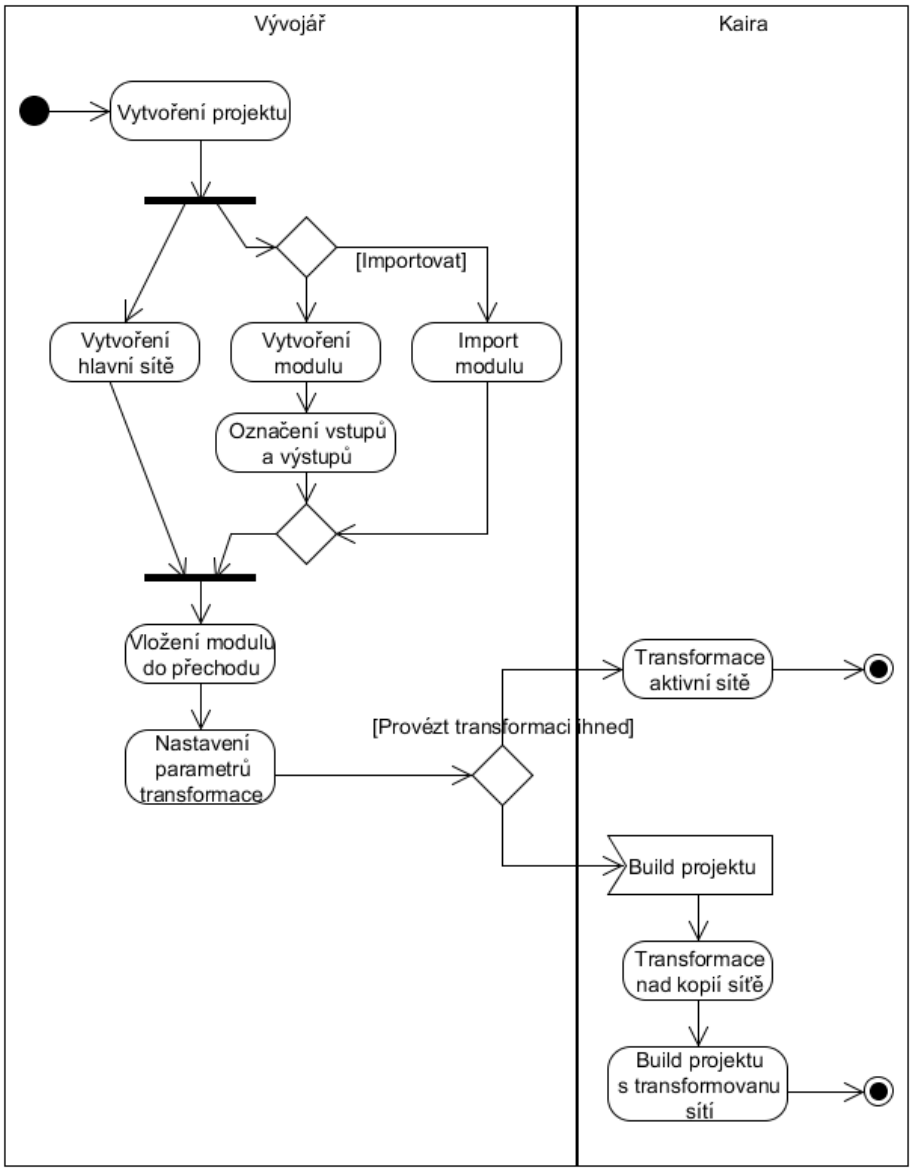
dalších parametrů. K dialogu se vrátím po vysvětlení automatických transformací v kapitole 3.2. Údaje není třeba zadávat ihned, k dialogu se lze vrátit kdykoliv znovu. Po vybrání modulu se název přechodu automaticky změní na název modulu a v kontextovém menu přechodu přibudou další dvě možnosti.

Rozšířené menu při pravém kliku na přechod tedy obsahuje tyto funkce:

- Set module - Volba podsítě nebo-li modulu a případný návrat k původnímu přechodu
- Insert subnet now - Umožňuje provézt celou transformaci okamžitě a výsledek vidět v editoru. Provede se totéž co by se provedlo automaticky v rámci buildu. Přechod zanikne a bude nahrazen podsítí a dalšími prvky, které budou zajišťovat požadované chování. Tahle položka vznikla úplně jako první, a to pro účel testování, ale není důvod proč jí v menu neponechat.
- Edit parameters - Již zmiňovaná funkce, která umožní kdykoliv editovat parametry propojení a transformace. V případě pozdější editace hlavní sítě i modulu je pravděpodobné, že bude nutné upravit i tyto parametry.

Celý postup pro vytváření modulu a jejich vkládání do přechodu je znázorněn na aktivitním diagramu (obr. 7). Prostředí pro vytváření sítí i modulů (vrchní část diagramu) již je v Kaiře implementováno. Rozšíření se bude týkat podpory vkládání modulu do

přechodu, to je kontextové menu přechodu a dialog pro zadávání parametrů. Hlavní část je automatická transformace, která zajistí správně chování modulu vůči hlavní síti a jejíž výsledkem bude síť, která se skládá pouze ze základních prvků jako jsou přechody, místa a hrany.

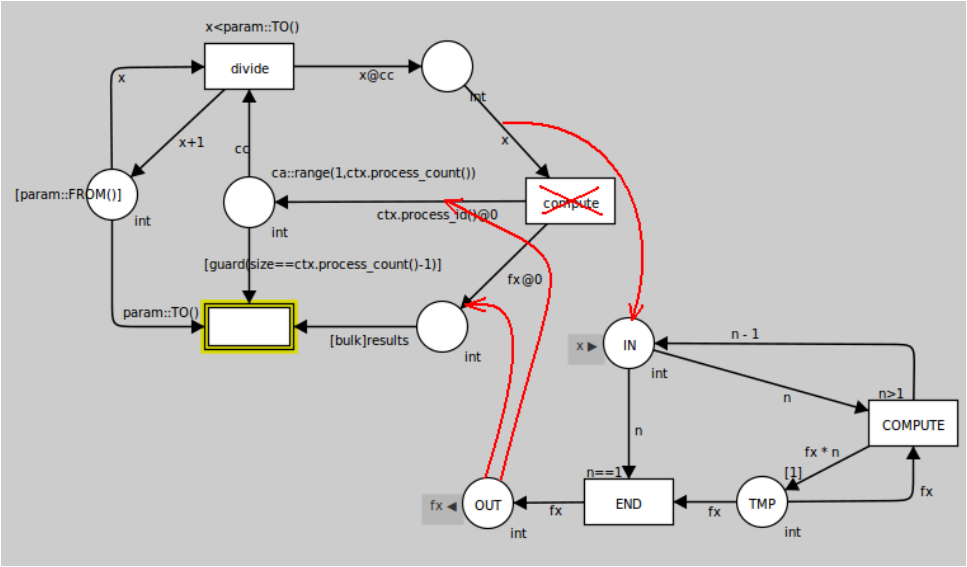


Obrázek 7: Diagram aktivit spojený s rozšířením Kairy o trasformace

3.2 Transformace podsítě

K uvědomění co bude automatická transformace muset obsahovat jsem zde vytvořil ilustraci (obr. 8). Na obrázku je vlevo hlavní síť. Jedná se o klasický příklad rozdělení úlohy metodou master-worker obdobně jako obr. 2 v textu výše. Pro zjednodušení však jedna úloha představuje výpočet pouze nad jedním číslem. Výpočet by měl proběhnout v přechodu `compute`, ale ten je přeškrtnutý, protože právě tento přechod bude nahrazen modulem na obrázku vpravo. V následujícím textu je tento přechod označován jako „původní přechod".

Vstupem modulu je jedno přirozené číslo `x` typu integer. Modul vypočítá faktoriál tohoto čísla a předá je to výstupního místa, kde výstup je označen jako `fx`. Síť funguje tak, že při každém provedení přechodu `COMPUTE` se mezivýsledek uložený v místě `TMP` přenásobí číslem n. Nazačátku obsahuje místo `TMP` jedničku, aby při násobení v prvním kroku zůstal mezivýsledek stejný. Číslo `n` je uchováváno ve vstupním místě `IN`, jehož vstupem bylo číslo x, takže na začátku platí, že n se rovná x. Postupně se pak nv každém kroku snižuje o 1. Ukončení výpočtu zajistí guard nad přechody `COMPUTE` a `END`. Výpočet se provádí pouze dokud je $n > 1$. V případě, že je $n = 1$ aktivuje se přechod `END` a přesune výsledek do výstupního místa `OUT`.



Obrázek 8: Ilustrace hlavní sítě a modulu, který mý být vložen na místo přechodu

Ve chvíli, kdy se výsledek zapíše do místa `OUT` by měl být přechod, ve kterém je modul vložený, považován jako provedený a evoluce sítě by měla pokračovat v hlavní síti. Co je potřeba přidat a transformovat se pokusím shrnout v bodech.

- Propojení obou sítí - Z ilustrace je zřejmé, že nebude stačit propojit oba moduly pouze hranami. Červenou šipkou je naznačeno co s čím by mělo byt propojeno.

Původní přechod zanikne a bude nahrazen modulem, takže propojovat se bude místo s místem, což nástroj ani teorie o Petriho Sítích neumožňuje.

- Opakovatelnost - Při provádění modulu mohou v místech vznikat a zanikat tokeny. Po transformaci bude výsledkem jedna síť a je potřeba zajistit aby se ta část sítě, která vznikne nahrazením přechodu modulem byla za běhu programu schopna vrátit do původního stavu. V opačném případě může být výsledek při každém opětovném provádění modulu ovlivněn předchozím provedením.
- Vzájemné umístění - V nabídce existuje možnost viditelného provedení transformace. Modul je v editoru nepochybně větší než přechod, který má nahradit, proto bude nutné řešit rozmístění nově vložených objektů. Rozmístění by bylo třeba řešit i kdyby tato možnost v nabídce neexistovala, protože v síti se kromě míst, přechodů a hran mohou vyskytovat ještě objekty na pozadí zvané init-area, které budou upřesněny později.

3.2.1 Propojení sítí

Vezmeme-li v úvahu, že přechod bude nahrazen modulem, pak se modul bude muset propojovat s místy, která byla původně propojena s přechodem. Rozhraní modulu také začíná a končí místy. K řešení tohoto problému se naskytují dvě možnosti.

1. Sjednotit místa v jedno
2. Vložit mezi místa přechod

První možnost je spíše minimalistická s důrazem na výkon. Sjednotit by se mohlo vstupní místo modulu s místem, ze kterého vedla hrana do původního přechodu. Oba dva místa by vždy měla být stejného datového typu. Sjednotit lze také výstupní místo s místem, do kterého vedla hrana z původního přechodu. Síť bude z pohledu programu po takovém spojení v pořádku, stačí převézt všechny hrany z jednoho místa do druhého a pak to první smazat. Složitější ale bude zajistit aby se modul choval jako přechod. Provedení přechodu je elementární operace a musí se provézt vcelku. V jednom kroku musí vzít tokeny ze vstupních míst, provézt co je jeho úlohou a vytvořit tokeny ve výstupních místech. Provedení modulu není elementární operace a má-li se navenek chovat jako přechod, je potřeba zajistit aby do procesu nebyly přibírány nové tokeny ze vstupních míst a aby během procesu nebylo možné odebrat tokeny ze vstupních míst. Místa jsou pasivní prvky a veškerá manipulace s tokeny je řízena přechody, a proto se bez přidání přechodu neobejdeme. Navíc není kam umístit podmínku(guard), kterou mohl obsahovat původní přechod.

Vhodnější způsob je druhá možnost, vložit místa mezi přechody. Úkolem přechodů bude uvézt modul do dvou různých stavů.

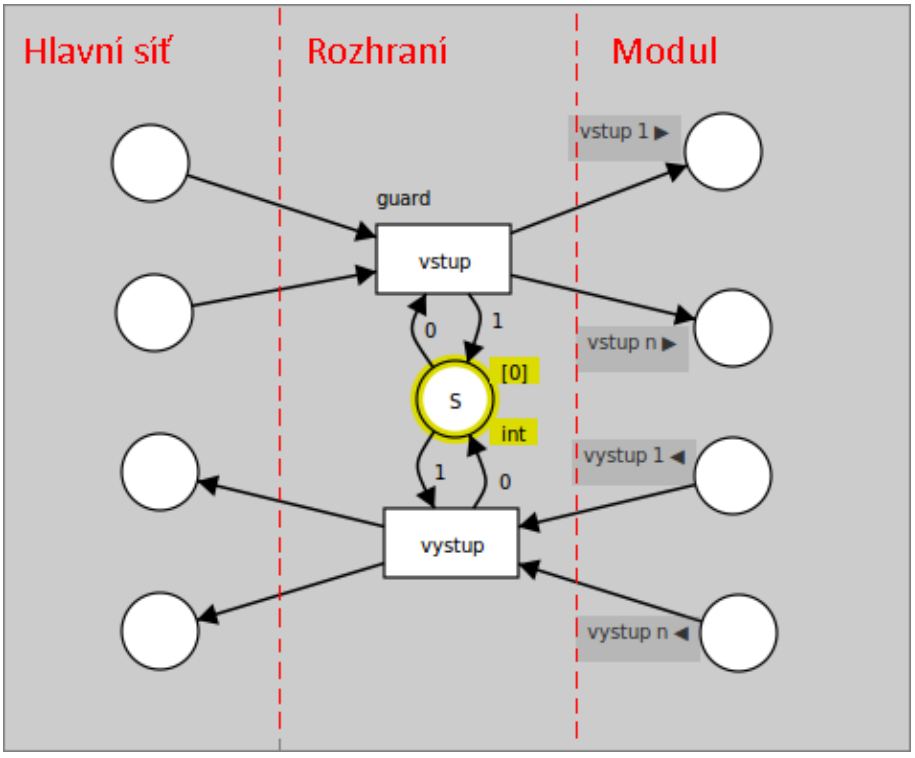
1. Stav volný - Je to prvotní stav a pokud jsou k dispozici vstupní tokeny, lze modul spustit

2. Stav obsazený - Stav kdy byl modul spuštěn a provádí se. Modul ve stavu setrvá dokud jsou uvnitř modulu proveditelné přechody. Když v modulu není žádný proveditelný přechod lze přejít zpět do prvotního stavu „volný“.

Konstrukce, která zajistí tyto dva stavy je na obrázku 9. K realizaci dvou stavů jsou potřeba dva přechody. Jeden přechod je nazván vstupní a jeho provedením se dostanou tokeny z vnější sítě do vstupních míst modulu. Modul přejde do stavu „obsazený“. Provedením druhého přechodu, výstupního, jsou všechny tokeny z výstupních míst modulu převedeny do vnější sítě a modul přejde do stavu „volný“.

Vstupní přechod bude přebírat guard původního přechodu, protože guard přechodu se vztahuje vždy na vstupní tokeny a má vliv na to zda je přechod proveditelný nebo ne. Vstupní i výstupní přechod přebírají také prioritu původního přechodu. Všechny ostatní přechody v modulu budou mít svou prioritu zvětšenou minimálně o jeden stupeň, tak aby měly větší prioritu než výstupní přechod. Tím se zajistí, že výstupní přechod se neprovede dokud může pokračovat evoluce modulu.

Místo `S` uchovává informaci o stavu formou `0` když je modul ve stavu „volný“ a formou `1` když je přechod v druhém stavu. Vstupní přechod by se nikdy neměl provézt podruhé dokud nebyl proveden výstupní přechod. Hrany mezi místem `S` a přechody jsou popsány tak, aby se přechody prováděli střídavě.



Obrázek 9: Rozhraní mezi modulem a hlavní síťí

Po přidání přechodů na rozhraní mezi hlavní sítí a modulem zbývá ještě propojit přechody a místa hranami. Hrany vedoucí z vnější sítě do přechodu s vloženým modulem již svůj popisek mají a budou pouze přesměrovány na vstupní přechod. Podobně budou hrany vedoucí z přechodu s vloženým modulem přesměrovány tak, aby vedli od výstupního přechodu. Popisek však zatím neexistuje pro hrany propojující přechody na rozhraní se vstupními a výstupními místy v modulu. Často bude možné popisek doplnit automaticky, například když má modul pouze jeden vstup a jeden výstup. V některých případech však bude existovat více různých možností a k popsání hran bude zapotřebí interakce uživatele. Uživatel bude mít k dispozici dialog jako je na obrázku 6 v předcházející kapitole.

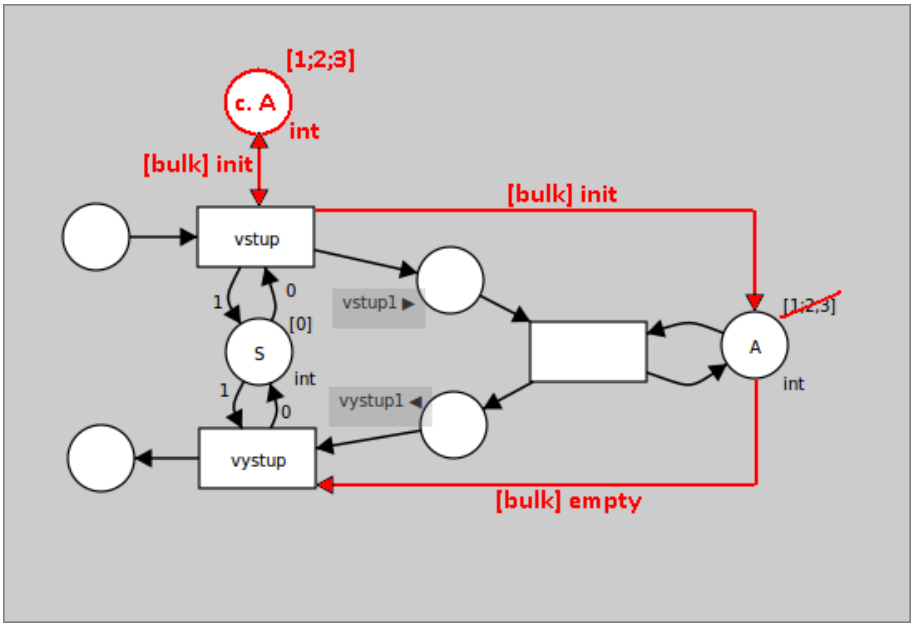
Původně jsem dialog navrhnul tak, aby pod sebou zobrazil seznam hran, které se týkají přechodu s vloženým modulem. Úkolem uživatele mělo být vybrání vstupního nebo výstupního místa ze seznamu pro každou zobrazenou hranu. Popisek měl být pouze zkopírován. Tato metoda nefungovala, protože hrany mohou obsahovat i složitější konstrukci nebo funkci a kopírovat funkci, tak aby byla na hraně vedoucí do přechodu i z přechodu by nedávalo smysl. Dialog má nakonec takovou podobu, aby zobrazil všechny důležité okolnosti na jednom místě a vyzval uživatele k zadání popisku příslušné hrany přímo.

3.2.2 Navrácení sítě do původního stavu

Během každého spuštění modulu by měl být jeho počáteční stav stejný. Stav sítě je dán obsahem míst. Cílem další transformace je tedy zajistit při každém spuštění modulu totožné rozložení tokenů v místech. Zapotřebí bude manipulace s tokeny. Konkrétně odstranění nově vzniklých tokenů a obnovení původních tokenů. Manipulovat s tokeny lze pouze prostřednictvím přechodů a nejvhodnější je použít vstupní a výstupní přechod. Uvažoval jsem nad dvěma způsoby jak docílit uvedení modulu do původního stavu po jeho provedení.

1. Ke každému místu v modulu se vytvoří jeho přesná kopie, včetně řetězce nebo kódu pro inicializaci dat. Původním místům bude řetězec nebo kód smazán. Při provedení vstupního přechodu budou data z nových míst okopírována do původních míst a modul se tak dostane do původního stavu. Provedením výstupního přechodu budou data ze všech míst kromě nově vytvořených kopií vybrána a zahozena. Příklad je na obrázku 10

2. Druhou možností je vytvořit prázdné kopie všech míst, které ale musí být stejného datového typu. Provedením vstupního přechodu budou data z původních míst zkopírována do nových prázdných míst sloužících jako záloha. Provedením výstupního přechodu budou data z původních míst zahozena a přesunou se do nich data uložená v záložních místech. Narozdíl od první varianty, kde jsou inicializační data zálohována pevně předem, tahle varianta zálohuje data při spuštění modulu a později musí data ze zálohy obnovit takže obsahuje ke každému místu jednu hranu navíc. Pro lepší přehlednost výsledné sítě jsem zvolil první variantu.



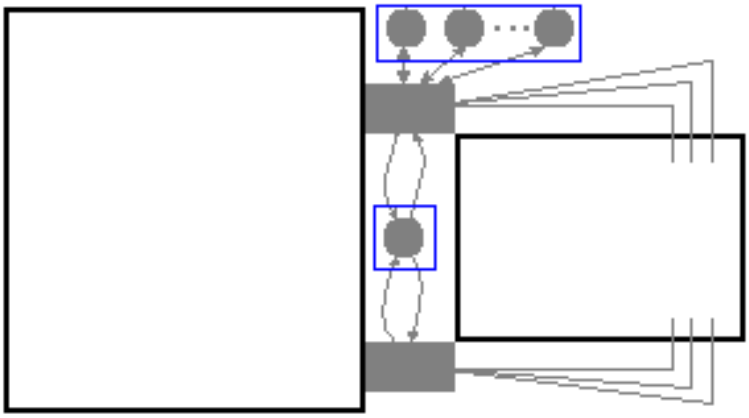
Obrázek 10: Konstrukce pro vrácení sítě do původního stavu

3.2.3 Umístění

Důvodem k řešení rozmístění je hlavně možnost testování a funkce pro viditelnou transformaci. Dalším nezanedbatelným důvodem je však jeden z konstrukčních prvků nástroje zvaný init-area. Jedná se o modrou obdélníkovou oblast umístěnou na pozadí sítě s jedním popiskem, který obsahuje výčet nebo rozsah čísel procesů. Místa nacházející se uvnitř oblasti budou inicializována na všech procesorech uvedených v popisku init-area. Místa, která nejsou v init-area jsou vždy inicializovány pouze v procesu s číslem 0 a v ostatních procesech jsou prázdné. Při vkládání modulu do hlavní sítě se tedy žádné místo nesmí dostat do oblasti init-area, do které nepatří.

Init-area bude použito také v transformaci. Jedno init-area bude pod všemi nově přidanými místy určenými pro zálohu inicializačních hodnot. Druhé init-area bude pod místem mezi vstupním a výstupním přechodem, které zajišťuje jejich střídání. Obě init-area budou mít v popisku celý rozsah dostupných procesů. To nám zajistí, že modul se bude chovat správně ve všech procesech. Vše je znázorněno na obrázku 11.

Na obrázku jsou obdélníkem naznačeny hranice hlavní sítě vlevo a hranice modulu vpravo. Světlejší barvou je znázorněna celá přidaná konstrukce. Modul je zarovnán vedle hlavní sítě napravo nahoře, protože podobně jako při psaní textu je většina modelů tvořena zleva doprava a od vrchu dolů. Mezi hlavní síti a modulem se nachází vstupní a výstupní přechod. Nad vstupním přechodem jsou ve vodorovné řadě umístěny záložní kopie míst, které je potřeba při každém spuštění inicializovat. Hrany pro inicializaci a vyprázdnění míst jsou vedeny od vstupního nebo výstupního přechodu vodorovně a směr změní až



Obrázek 11: Rozmístění objektů v síti se siluetou přidané konstrukce

nad cílovým místem, aby co nejméně překrývali modul. Navíc jsou hrany od sebe mírně odkloněny, aby nesplynuly v jednu.

4 Poznámky k implementaci

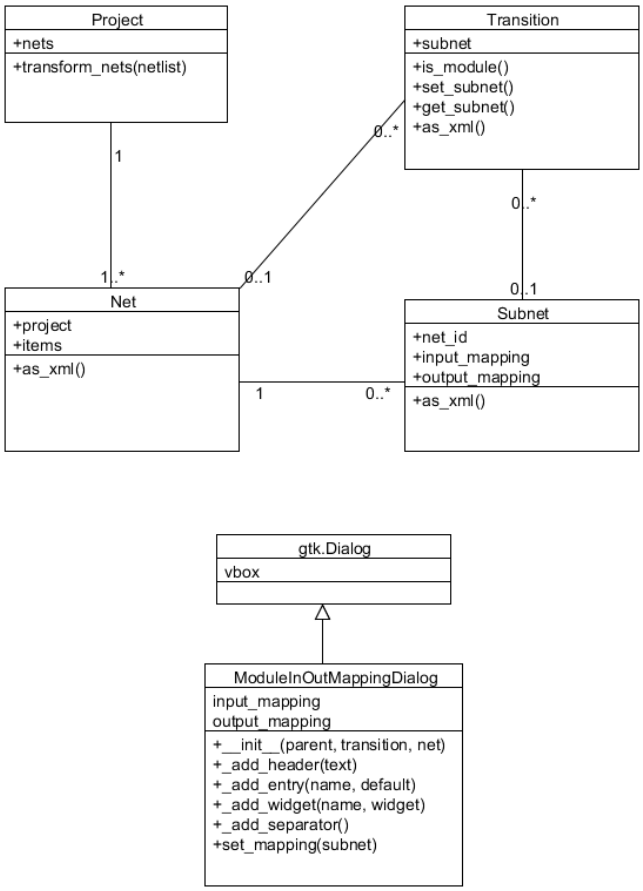
4.1 Technické parametry a použité technologie

Kaira je vyvíjena pro systémy založené na GNU/Linux. Implementace a odzkoušení příkladů proběhlo na platformě Ubuntu.

Programuji pouze v oblasti uživatelského rozhraní. Jako programovací jazyk pro uživatelské rozhraní včetně tvorby modelů je použit jazyk Python [8], verze 2.7.

K demonstraci a odzkoušení řešení v kapitole 5 je použita knihovna Octave [9], verze 3.6.1. Z implementací MPI je použito MPICH2.

4.2 Třídní diagram



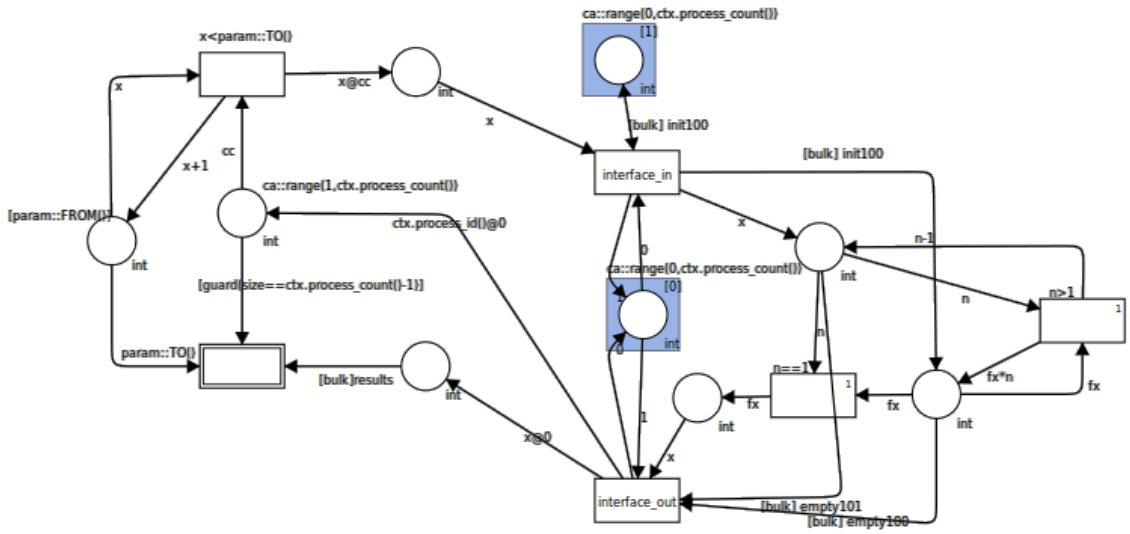
Obrázek 12: Třídní diagram: dialog k transformacím a relace podsítě

Třídní diagram 12 zobrazuje vztah přechodu, hlavní sítě a podsítě. Nově implementované jsou třídy Subnet a ModuleInOutMappingDialog. Subnet je třída, která má za úkol udržovat informace o transformaci a obsahuje pouze id skutečné podsítě. ModuleInOutMappingDialog je snadno rozšiřitelný dialog pro zadání parametrů transformace a jeho rodičovskou třídou je Dialog z balíčku gtk[8]. Samotná transformace není metodou žádné třídy. Je to skript, který je volán editorem nebo třídou Project při sběru informací pro build.

Řešení je implementováno v samostatné větvi verzovacího systému git.

5 Ukázka řešení

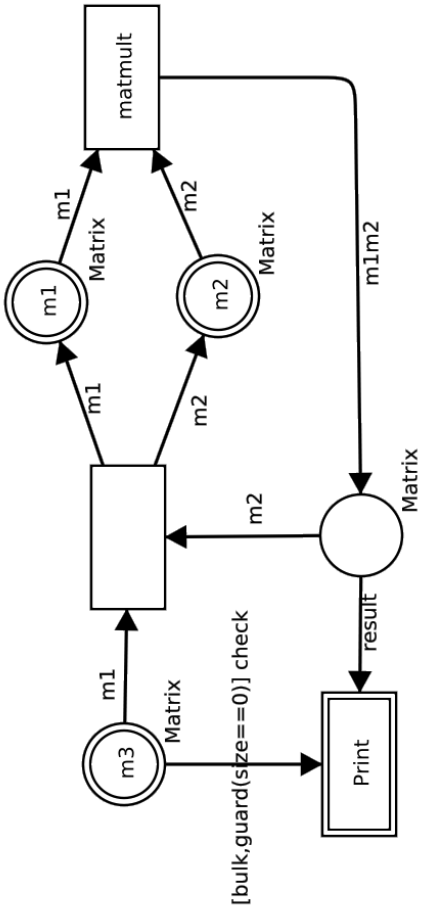
První ukázka bude bez komentáře. Jedná se o příklad, který byl již v textu použit a na obrázku 13 je tento příklad po automatické transformaci.



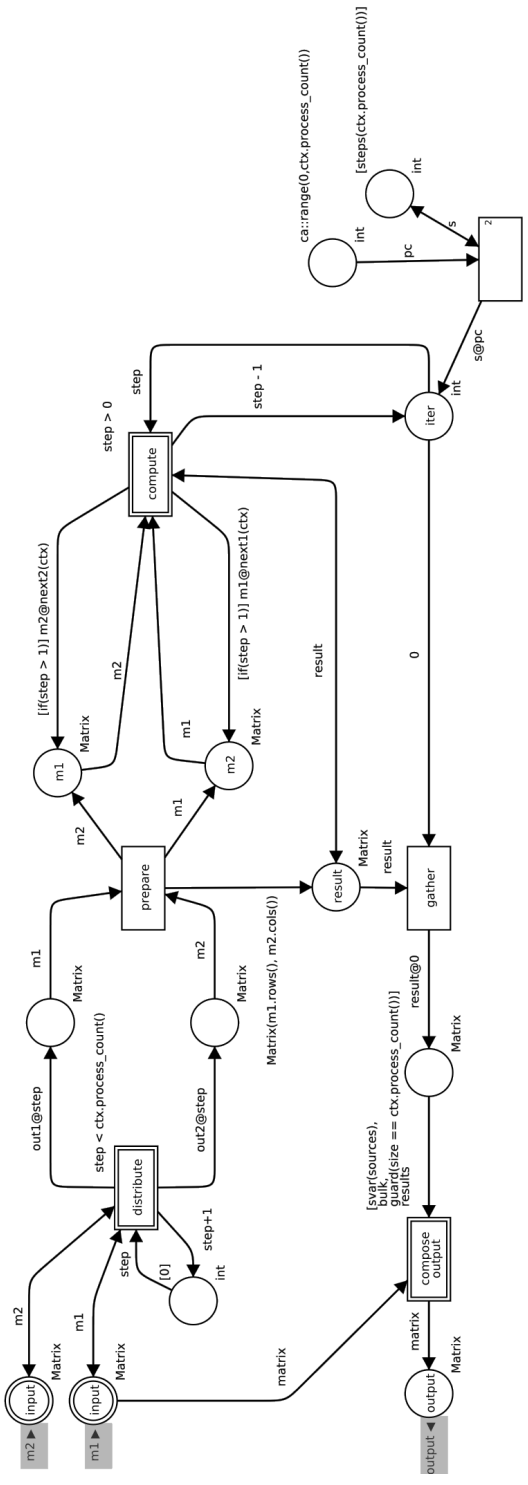
Obrázek 13: Ukázka transformované sítě z obrázku 8

Následující příklad je již složitější a praktičtější. Modulem, který se bude vkládat do přechodu je síť provádějící násobení dvou matic paralelně a cílem je použít tento modul k násobení tří matic. Na obrázku 14 je hlavní síť, na obrázku 15 je modul a na posledním obrázku č. 16 je celá síť po automatické transformaci. Modul je v hlavní síti vložen do přechodu s názvem `matmult`.

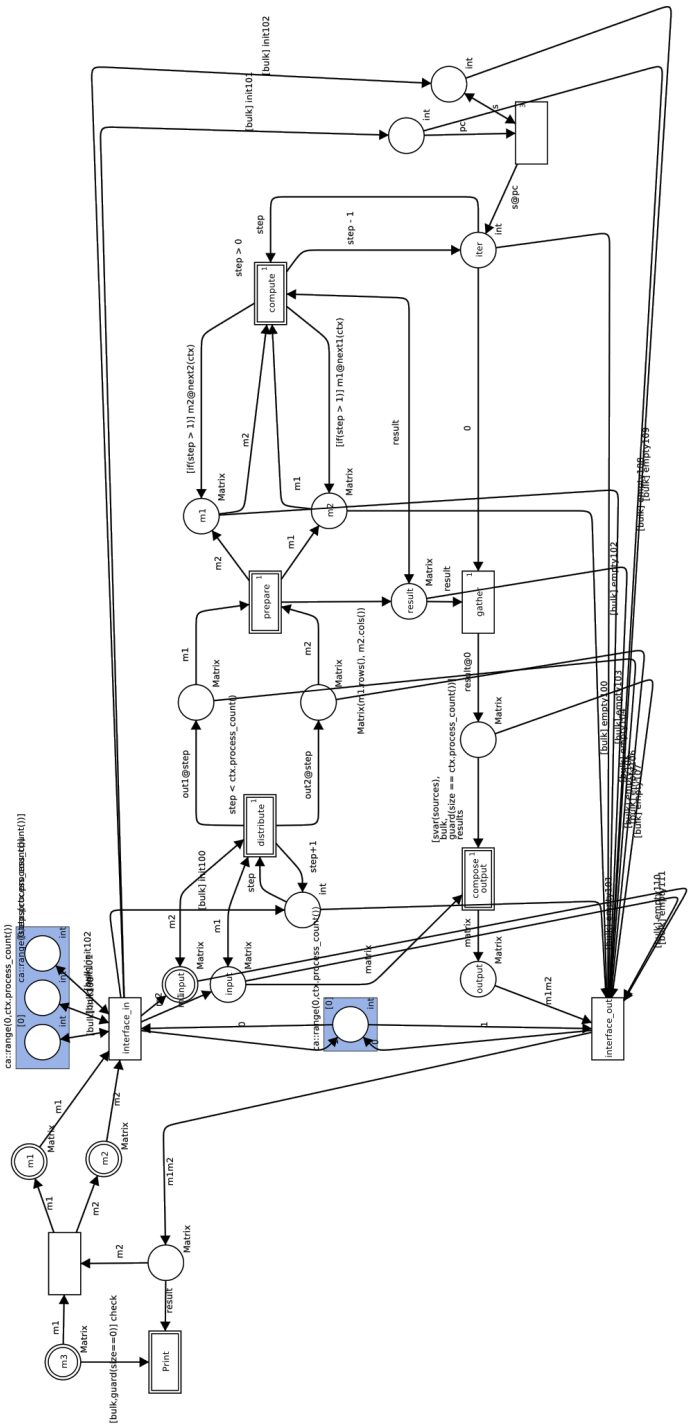
Program je kompilovatelný a spustitelný. Výsledek násobení obsahoval chybu, takže modul násobení dvou matic je třeba ještě odladit. Problém je pravděpodobně se synchronizací dílčích výpočtů, protože ve výsledcích se opakují stejné bloky matic, ale mění se jejich pozice. Automatická transformace však zafungovala správně.



Obrázek 14: Násobení tří matic před transformací



Obrázek 15: Modul násobení matic



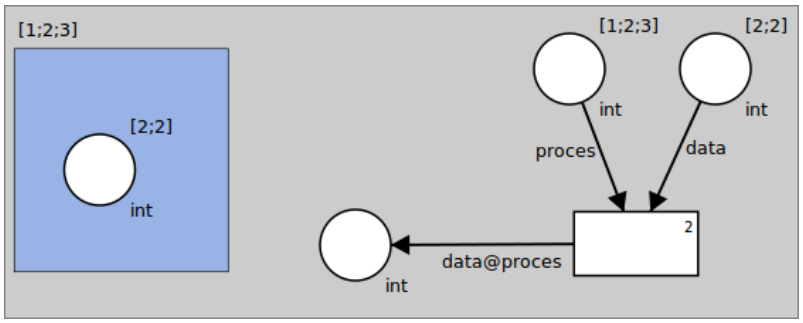
Obrázek 16: Násobení tří matic po transformaci

6 Další transformace

Příklady, na kterých bylo demonstrováno řešení jsou svým způsobem specifické. První příklad (obr. 13) řešil distribuci úlohy metodou master-worker, kde výpočetní úloha byla realizována modulem. Modul uvnitř neobsahoval žádnou práci s procesory, takže prováděl sekvenční výpočet, ale za to byl proveditelný na libovolném procesoru. Chování modulu z pohledu hlavní sítě pak ve výsledku souhlasilo s chováním přechodu. V druhém příkladu, který řešil násobení třech matic (obr. 16) už modul obsahoval práci s procesory. V tomto modulu však bylo určeno, že distribuci a konečný sběr výsledků provádí proces s číslem 0. Navíc modul nezanechával v jiných procesech v místech žádné tokeny. Hlavní síť byla také inicializovaná a prováděna pouze v procesu 0, takže chování modulu z pohledu hlavní sítě se nelišilo oproti chování přechodu.

Dá se říct, že transformovaná síť je plně funkční v případě, kdy modul neodesílá data na jiný proces, a nebo v případě, kdy modul využívá další procesy, ale v místech nezanechá přebytečné tokeny. Oba dva případy nejsou nijak vyjímečné, a vždy je možné modul do jejich podoby upravit. Kdybychom však chtěli automatickou transformaci vylepšit, první věc co bude na seznamu je rozšíření schopnosti navrácení sítě do původního stavu. Rozšíření spočívá v tom, že po skončení modulu by byla sesbírána data ze všech procesů, na které modul odesílal data. Řešit by se musela také opětovná inicializace dat.

Díky nedávnému zavedení priorit pro přechody je možné inicializaci a sběr dat na více procesech vyřešit relativně jednoduchou konstrukcí. Inicializace dat na více procesorech nebo na jiném procesu než procesu s číslem 0 se v kaiře modeluje pomocí init-area. Init-area je možné nahradit konstrukcí jako je na obr. 17. Init-area je nahrazeno dvěma místy a přechodem. Jedno místo obsahuje inicializační data a druhé místo obsahuje výčet procesů, na kterých mají být data inicializována. Opětovnou inicializaci těchto míst při každém spuštění modulu již zajistí stávající konstrukce. Data pak jsou rozeslána prostřednictvím přechodu. Tento přechod bude muset mít nejvyšší prioritu v modulu, aby se inicializace provedla po spuštění modulu ze všeho nejdříve.



Obrázek 17: Náhrada init-area s pomocí míst a přechodů

Zahození redundantních dat z procesů bude komplikovanější. Data musí zahodit každý proces zvlášť, ale při tom musí všechny procesy vědět, kdy provádění modulu skončilo. Řešením by mohlo být například bezprostředně před výstupní přechod zařadit

článek, který všem procesům vyšle signál o ukončení provádění modulu. Tento signál bude podmínkou proveditelnosti přechodu, který zahodí přebytečná data ze všech míst, podobně jako ve stávajícím řešení. Pro rozeslání signálu může sloužit stejná konstrukce, jako byla na obrázku 17. Signál bude muset být rozesílán všem procesům, protože není možné předem zjistit, které procesy budou do provádění modulu zapojeny a rozeslání signálu bude méně náročná operace než zaznamenávat cílový proces pro zprávy v každém přechodu.

Dalším krokem k lepšímu využití transformací by mohlo být rozšíření syntaxe na hranách tak, aby bylo možné namísto konkrétního čísla procesu zadávat proměnnou. Moduly pak mohou být méně konkrétní. Například modul řešící násobení matic(obr. 15) odesílá výsledky vždy na proces s číslem 0. Číslo procesu, který sbírá výsledky lze změnit, ale vždy je to jen jedno číslo, a proto nemá smysl modul spouštět paralelně na více procesech. Jelikož je modul vkládán do přechodu, spuštění modulu probíhá vždy na procesu, na kterém byli splněny všechny podmínky pro provedení přechodu. Stejný proces také zajišťuje vygenerování výstupních dat. Tento proces lze označit jako řídící proces modulu a v již existujících modulech je to často proces s číslem 0. Rozšíření by mohlo vypada tak, že číslo procesu 0 bychom nahradili proměnnou textttw a k výpočtu by se použily procesy w až $w + o$. Proměnnáw je číslo řídícího procesu, na který přijímá vstupní tokeny a odesílá výstupní tokeny. Proměnná o určuje počet procesů následujících za w, které budou ve výpočtu použity. Konstrukce která připraví výčet všech dostupných procesů, jako je $range(0, ctx.process_count())$ by byla nahrazena konstrukcí $range(w, w+o)$. Číslo řídícího procesu w je v modulu vždy známo a číslo o bude zadáváno jako parametr transformace. Proměnné číslo procesu lze využít v příkladu rozdělení úlohy metodou master-worker. Master by úlohu rozděloval a odesílal na procesy s offsetem. Řekněme že offset bude 4. Potom master rozdělí úlohu procesům 0, 4, 8 a tak dále. Tento offset bude zadán modulu jako parametr o, takže modul může k výpočtu využít vždy 4 po sobě jdoucí procesy. Pro implementaci transformace modulu bude stačit číslo w a o uchovávat v místech. Přechod vyskytující se uvnitř modulu, který bude na hraně nebo v guardu obsahovat jedno z těchto čísel, jej může získat z místa, ale musí jej do místa nazpět vrátit.

7 Shrnutí

Cílem mé práce bylo pokusit se obnovit možnost využívat moduly při tvorbě modelu v nástroji Kaira. Dřívější řešení popsané v [4] bylo implementováno hlavně v oblasti generování kódu. Moduly byly v modelu vkládány do přechodů, protože obvykle plní nějakou funkci, přebírají vstupy a generují výstup a tím se nejvíce přibližují významu přechodu. Za běhu programu se při provádění přechodu vytvořila nová samostatná instance modulu. Instance modulu běžela dokud v ní byl proveditelný některý přechod. Po skončení modulu byly z označených výstupních míst převzaty tokeny a tím provádění přechodu v hlavní síťi skončilo. Řešení umožňovalo modul spouštět opakovaně a důkladně ho ukončit, ale v praktickém testování se ukázalo, že instance modulu se vytvářejí spíše opakovaně na jednom procesu, než rovnoměrně mezi procesy. Jelikož vše řeší generátor kódu, nebylo ve fázi modelování dostatek prostředků jak zjistit a ovlivnit, na kterém procesu budou instance modulu vytvářeny. Novým záměrem je zachovat možnost vkládání modulu a implementovat transformaci, která vloží modul do hlavní sítě namísto přechodu a s použitím základních konstrukcí nástroje docílit toho, že se bude modul vůči hlavní síti chovat jako přechod. Stejnou problematiku řešil ve své absolventské práci v roce 2013 Lukáš Tomaszek[10]. Jeho řešení bylo na jednoduchých příkladech funkční, ale na praktických příkladech nefungovalo a transformovaná síť byla velice rozsáhlá na to, aby se v ní uživatel zorientoval. V poslední době bylo do Kairy zavedeno rozšíření umožňující nastavit prioritu přechodu a tím ovlivnit pořadí provádění přechodu. Zavedení priorit by mělo značně snížit objem transformace a nově přidaných prvků, a proto jsem transformace implementoval od začátku.

V rámci této práce jsem rozšířil uživatelské rozhraní nástroje Kaira pro podporu vkládání modulu do přechodu a provedení transformace na požadavek uživatele nebo automaticky před započetím buildování projektu. Transformace vloží modul do hlavní sítě a síť předělá tak, aby se modul vůči zbytku sítě choval jako jeden přechod. S pomocí nastavení priorit přechodům se mi podařilo docílit toho, aby průběh modulu nebyl narušen vstupem dalších dat a skončil tehdy, když není uvnitř modulu proveditelný žádný přechod. Všechna místa uvnitř modulu se vždy po skončení evoluce uvnitř modulu vrátí do původního stavu. Po transformaci je celá síť, až na několik hran protínajících místo nebo přechod, přehledná a lze v ní rozpoznat modul i zbytek sítě před transformací, takže uživatel může transformovanou síť simulovat a upravovat. Součástí implementace je také dialog pro zadání parametrů transformace. Tento dialog zatím obsahuje pouze formulář pro zadání údajů k propojení vstupních a výstupních míst modulu se zbytkem sítě.

Implementované řešení umožňuje spustit modul na libovolném procesu a je schopno uvést všechna místa v modulu na tomto procesu do původního stavu, takže se modul chová vůči hlavní síti jako přechod. Řešení jsem otestoval na příkladu, kde je modul řešící paralelní násobení dvou matic použit dvakrát po sobě, aby mezi sebou vynásobil tři matice. Řešení se ukázalo jako funkční a může tak nahradit jinou variantu řešení, kterou je napsat zvláštní program využívajícího kód vygenerovaný z modulu.

Při využívání rozšíření si musí programátor pohlídat modul, aby nezůstávala data na jiném procesu, než na kterém je modul spuštěn. Moduly jsou často navrhovány tak, že se data ze vzdálených procesů sesbirají, ale není to pravidlem. Z tohoto důvodu jsem v ka-

pitole 6 popsal další možnosti transformace, které mohou být implementovány. Mezi tyto možnosti patří uvedení modulu do původního stavu na všech procesech při každém spuštění modulu a zavedení proměnných na místo konkrétních čísel procesu uvnitř modulu. Prvním předmětem budoucí práce by však mělo být doimplementováni simulace, kterou jsem ještě nepřipravil pro případ, kdy je modul skryt v přechodu. Možným řešením je při simulaci vytvořit také instanci modulu, která se bude simulovat zvlášť. Po každé, když uživatel klikne na přechod s modulem, přesunou se vstupní tokeny z toho přechodu do vstupních míst v simulaci modulu. Simulaci uvnitř modulu může uživatel ovládat sám, ale lze ji také provézt celou automaticky.

8 Reference

[1] K. Jensen, L. M. Kristensen *Coloured Petri Nets*, Springer Berlin Heidelberg, 2009. ISBN: 978-3-642-00283-0

[2] *MPI documents* [online], 2012-09-21 [vid. 2014-04-29]. Dostupné na: http://www.mpi-forum.org/docs/docs.html

[3] *Kaira, High-level tool for MPI* [online], 2014-03-19, [vid. 2014-04-29]. Dostupné na: http://verif.cs.vsb.cz/kaira

[4] S. Böhm, M. Běhálek, *Usage of petri nets for high performance computing*, Proceedings of the 1st ACM SIGPLAN workshop on Functional high-performance computing, ACM, New York, NY, USA, 2012, s. 37-48. Dostupné na: http://doi.acm.org/10.1145/2364474.2364481

[5] W. Reisig, G. Rozenberg, eds., *Lectures on Petri Nets I: Basic Models*, LNCS 149, Springer, 1998

[6] W. Reisig, G. Rozenberg, eds., *Lectures on Petri Nets II: Applications*, LNCS 1492, Springer, 1998

[7] J. Markl, *Hierarchické Petriho Sítě*, z: J.Markl, Petriho Sítě I, VŠB-Technická univerzita Ostrava. Ostrava, 2009, s. 20-22.

[8] P. Barry, *Head First Python*, O'Reilly Media, California, USA, 2010. ISBN: 978-1-4493-8267-4

[9] J. Schmidt Hansen, *GNU Octave*, Packt Publishing 2011. Ebook. ISBN:978-1-84951-333-3

[10] L. Tomaszek, *Implementace transformací modelů postavených na Petriho sítích v nástroji Kaira*, Bakalářská práce,VŠB-Technická univerzita Ostrava, Fakulta elektrotechniky a informatiky. Ostrava, 2013